42P17572 *PATENT*

UNITED STATES DESIGN PATENT APPLICATION

FOR

PROVIDING SUPPORT FOR SINGLE STEPPING A VIRTUAL MACHINE IN A VIRTUAL MACHINE ENVIRONMENT

Inventors:

STEVEN M. BENNETT

ANDREW V. ANDERSON

ERIK COTA-ROBLES

STALINSELVARAJ JEYASINGH

ALAIN KÄGI

GILBERT NEIGER

RICHARD UHLIG

SANJOY K. MONDAL

JASON BRANDT

Prepared by:

BLAKELY SOKOLOFF TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard, Seventh Floor
Los Angeles, CA 90025-1026

(408) 720-8300

EXPRESS MAIL CERTIFICATE OF MAILING

"Express Mail" mailing label number EV409361081US
Date of Deposit March 30, 2004
I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Mail Stop Patent Application, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450

Michelle Begay
(Typed or printed name of person mailing paper or fee)

Michelle Begay
(Signature of person mailing paper or fee)

PROVIDING SUPPORT FOR SINGLE STEPPING A VIRTUAL MACHINE IN A VIRTUAL MACHINE ENVIRONMENT

Field

[0001] Embodiments of the invention relate generally to virtual machines, and more specifically to providing support for single stepping a virtual machine in a virtual machine environment.

Background of the Invention

[0002] A single stepping technique is commonly used by debuggers in a conventional operating system (OS) environment. In particular, the single stepping technique is used to advance an application one instruction at a time, thus allowing the debugger to position the application at a particular point in the source code. Various mechanisms have been used to enable single stepping. For example, in the instruction set architecture (ISA) of the Intel® Pentium® 4 (referred to herein as the IA-32 ISA), a trap flag (TF) bit in the EFLAGS register is designated to allow a debugger to single step an application. When the TF bit is set to 1, a debug exception is generated following the completion of the next instruction. The debugger sets the TF bit after taking ownership of the debug exception (e.g., by creating a handler for the exception and assuring that the handler will be called in the event of a debug exception). However, the debug exception is generated only if the next instruction completes successfully. If the execution of the next instruction causes a fault (e.g., a page fault), the debug exception is not generated.

Rather, the exception is vectored, saving the values of EFLAGS register bits and clearing the TF bit. Upon completion of the handler, the saved value of the TF bit is restored, and the instruction is re-executed. If no faults occur during the re-execution, the debug exception is generated.

Brief Description of the Drawings

[0003] The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

[0004] Figure 1 illustrates one embodiment of a virtual-machine environment, in which the present invention may operate;

[0005] Figure 2 is a flow diagram of one embodiment of a process for providing support for single stepping a virtual machine in a virtual machine environment;

[0006] Figure 3 is a flow diagram of one embodiment of a process for responding to a request for a VM entry with a single stepping indicator set to a single stepping value;

[0007] Figure 4 is a flow diagram of one embodiment of a process for handling events having a higher priority than single stepping; and

[0008] Figure 5 is a flow diagram of an exemplary process for utilizing single stepping of a VM to facilitate virtualization of a device, according to one embodiment.

Description of Embodiments

[0009] A method and apparatus for providing support for single stepping a virtual machine in a virtual machine environment is described. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention can be practiced without these specific details.

[0010] Some portions of the detailed descriptions that follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer system's registers or memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of operations leading to a desired result. The operations are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

[0011] It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically

stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or the like, may refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer-system memories or registers or other such information storage, transmission or display devices.

[0012] In the following detailed description of the embodiments, reference is made to the accompanying drawings that show, by way of illustration, specific embodiments in which the invention may be practiced. In the drawings, like numerals describe substantially similar components throughout the several views. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention. Other embodiments may be utilized and structural, logical, and electrical changes may be made without departing from the scope of the present invention. Moreover, it is to be understood that the various embodiments of the invention, although different, are not necessarily mutually exclusive. For example, a particular feature, structure, or characteristic described in one embodiment may be included within other embodiments. The following detailed description is, therefore, not to be taken in a limiting sense, and the

scope of the present invention is defined only by the appended claims, along with the full scope of equivalents to which such claims are entitled.

[0013] Although the below examples may describe single stepping of a virtual machine in the context of execution units and logic circuits, other embodiments of the present invention can be accomplished by way of software. For example, in some embodiments, the present invention may be provided as a computer program product or software which may include a machine or computer-readable medium having stored thereon instructions which may be used to program a computer (or other electronic devices) to perform a process according to the present invention. In other embodiments, steps of the present invention might be performed by specific hardware components that contain hardwired logic for performing the steps, or by any combination of programmed computer components and custom hardware components.

[0014] Thus, a machine-readable medium may include any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer), but is not limited to, floppy diskettes, optical disks, Compact Disc, Read-Only Memory (CD-ROMs), and magneto-optical disks, Read-Only Memory (ROMs), Random Access Memory (RAM), Erasable Programmable Read-Only Memory (EPROM), Electrically Erasable Programmable Read-Only Memory (EEPROM), magnetic or optical cards, flash memory, a transmission over the Internet, electrical, optical, acoustical or other forms of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.) or the like.

[0015] Further, a design may go through various stages, from creation to simulation to fabrication. Data representing a design may represent the design in a number of manners. First, as is useful in simulations, the hardware may be represented using a hardware description language or another functional description language. Additionally, a circuit level model with logic and/or transistor gates may be produced at some stages of the design process. Furthermore, most designs, at some stage, reach a level of data representing the physical placement of various devices in the hardware model. In the case where conventional semiconductor fabrication techniques are used, data representing a hardware model may be the data specifying the presence or absence of various features on different mask layers for masks used to produce the integrated circuit. In any representation of the design, the data may be stored in any form of a machine-readable medium. An optical or electrical wave modulated or otherwise generated to transmit such information, a memory, or a magnetic or optical storage such as a disc may be the machine readable medium. Any of these mediums may "carry" or "indicate" the design or software information. When an electrical carrier wave indicating or carrying the code or design is transmitted, to the extent that copying, buffering, or re-transmission of the electrical signal is performed, a new copy is made. Thus, a communication provider or a network provider may make copies of an article (a carrier wave) embodying techniques of the present invention.

[0016] **Figure 1** illustrates one embodiment of a virtual-machine environment 100, in which the present invention may operate. In this embodiment, bare platform hardware 116 comprises a computing platform, which may be capable, for example, of executing a standard operating system (OS) or a virtual-machine monitor (VMM), such as a VMM 112.

[0017] The VMM 112, though typically implemented in software, may emulate and export a bare machine interface to higher level software. Such higher level software may comprise a standard or real-time OS, may be a highly stripped down operating environment with limited operating system functionality, may not include traditional OS facilities, etc. Alternatively, for example, the VMM 112 may be run within, or on top of, another VMM. VMMs may be implemented, for example, in hardware, software, firmware or by a combination of various techniques.

[0018] The platform hardware 116 can be of a personal computer (PC), mainframe, handheld device, portable computer, set-top box, or any other computing system. The platform hardware 116 includes a processor 118 and memory 120.

[0019] Processor 118 can be any type of processor capable of executing software, such as a microprocessor, digital signal processor, microcontroller, or the like. The processor 118 may include microcode, programmable logic or hardcoded logic for performing the execution of method embodiments of the present invention. Although **Figure 1** shows only one such processor 118, there may be one or more processors in the system.

[0020] Memory 120 can be a hard disk, a floppy disk, random access memory (RAM), read only memory (ROM), flash memory, any combination of the above devices, or any other type of machine medium readable by processor 118. Memory 120 may store instructions and/or data for performing the execution of method embodiments of the present invention.

[0021] The VMM 112 presents to other software (i.e., "guest" software) the abstraction of one or more virtual machines (VMs), which may provide the same or different abstractions to the various guests. **Figure 1** shows two VMs, 102 and 114. The guest software running on each VM may include a guest OS such as a guest OS 104 or 106 and various guest software applications 108 and 110. Each of the guest OSs 104 and 106 expect to access physical resources (e.g., processor registers, memory and I/O devices) within the VMs 102 and 114 on which the guest OS 104 or 106 is running and to perform other functions. For example, the guest OS expects to have access to all registers, caches, structures, I/O devices, memory and the like, according to the architecture of the processor and platform presented in the VM. The resources that can be accessed by the guest software may either be classified as "privileged" or "non-privileged." For privileged resources, the VMM 112 facilitates functionality desired by guest software while retaining ultimate control over these privileged resources. Non-privileged resources do not need to be controlled by the VMM 112 and can be accessed by guest software.

[0022] Further, each guest OS expects to handle various fault events such as exceptions (e.g., page faults, general protection faults, etc.), interrupts

(e.g., hardware interrupts, software interrupts), and platform events (e.g., initialization (INIT) and system management interrupts (SMIs)). Some of these fault events are "privileged" because they must be handled by the VMM 112 to ensure proper operation of VMs 102 and 114 and for protection from and among guest software.

[0023] When a privileged fault event occurs or guest software attempts to access a privileged resource, control may be transferred to the VMM 112. The transfer of control from guest software to the VMM 112 is referred to herein as a VM exit. After facilitating the resource access or handling the event appropriately, the VMM 112 may return control to guest software. The transfer of control from the VMM 112 to guest software is referred to as a VM entry.

[0024] In one embodiment, the processor 118 controls the operation of the VMs 102 and 114 in accordance with data stored in a virtual machine control structure (VMCS) 124. The VMCS 124 is a structure that may contain state of guest software, state of the VMM 112, execution control information indicating how the VMM 112 wishes to control operation of guest software, information controlling transitions between the VMM 112 and a VM, etc. The processor 118 reads information from the VMCS 124 to determine the execution environment of the VM and to constrain its behavior. In one embodiment, the VMCS is stored in memory 120. In some embodiments, multiple VMCS structures are used to support multiple VMs.

[0025] In one embodiment, the processor 118 includes single stepping logic 122 that receives a request of the VMM 112 to transfer control to the VM 102 or 114 (a request for a VM entry) and determines whether the VMM 122 has requested to single step the relevant VM. The single stepping functionality may have various uses in the virtual-machine environment 100. For example, the VMM 112 may request the single stepping functionality to debug system (i.e., privileged) or application code running in the VM 102 or 114. In another example, the VMM 112 may request the single stepping functionality to achieve virtualization of a particular device or platform during the operation of the VM 102 or 114 (e.g., a memory-mapped I/O device for which the VMM 112 wishes to allow the VM's read operations but not write operations). In yet another example, the VMM 112 may request the single stepping functionality to trace the execution of the VM 102 or 114 (e.g., to maintain the history of VM operation when testing the design of a new system).

[0026] The single stepping logic 122 determines whether the VMM 122 has requested the single stepping functionality based on a current value of a single stepping indicator. In one embodiment, the single stepping indicator is stored in the VMCS 124. Alternatively, the single stepping indicator may reside in the processor 118, a combination of the memory 120 and the processor 118, or in any other storage location or locations.

[0027] In one embodiment, the VMM 112 sets the value of the single stepping indicator before requesting a transfer of control to the VM 102 or 114.

Alternatively, each of the VMs 102 and 114 is associated with a different single stepping indicator that is set to a predefined value or changed during the life of the VM.

[0028] If the single stepping logic 122 determines, upon receiving the request of the VMM 112 for a VM entry, that the single stepping indicator is set to a single stepping value (e.g., 1), the single stepping logic 122 transitions control to the VM, executes the first VM instruction, and if the execution of the first VM instruction completes successfully, transfers control back to the VMM 112 (i.e., a VM exit is generated). In one embodiment, the single stepping logic 122 also notifies the VMM 112 (e.g., using a designated reason code) that the VM exit is caused by the current value of the single stepping indicator.

[0029] If the execution of the first VM instruction is unsuccessful (i.e., it causes a fault), the single stepping logic 122 determines whether the resulting fault is associated with a VM exit (e.g., by examining relevant execution control information in the VMCS 124 to determine whether it indicates that the fault requires a VM exit). If so, the single stepping logic 122 transfers control to the VMM 112 and, in one embodiment, notifies the VMM 112 that this VM exit is caused by the fault. If the fault does not require a VM exit, the single stepping logic 122 delivers the fault to the VM. In one embodiment, delivering of the fault involves searching a redirection structure for an entry associated with the fault being delivered, extracting from this entry a descriptor of the location of a routine designated to handle this fault, and jumping to the beginning of the routine using the descriptor. Routines

designated to handle corresponding interrupts, exceptions or any other faults are referred to herein as fault handlers.

[0030] During the delivery of a fault, the processor 118 may perform one or more address translations, converting an address from a virtual to physical form. For example, the address of the interrupt table or the address of the associated handler may be a virtual address. The processor may also need to perform various checks during the delivery of a fault. For example, the processor may perform consistency checks such as validation of segmentation registers and access addresses (resulting in limit violation faults, segment-not-present faults, stack faults, etc.), permission level checks that may result in protection faults (e.g., general-protection faults), etc.

[0031] Address translations and checking during fault vectoring may result in a variety of faults, such as page faults, general protection faults, etc. Some faults occurring during the delivery of a current fault may cause a VM exit. For example, if the VMM 112 requires VM exists on page faults to protect and virtualize the physical memory, then a page fault occurring during the delivery of a current fault to the VM will result in a VM exit.

[0032] In one embodiment, the single stepping logic 122 addresses the above possible occurrences of additional faults by checking whether the delivery of the current fault was successful. If the delivery of the current fault completes successfully, the single stepping logic 122 generates a VM exit prior to executing any instructions of the fault handler, and notifies the VMM 112 that the VM exit is caused by the current value of the single stepping

indicator. If the single stepping logic 122 determines that the delivery was unsuccessful, it repeats the above processing for the new fault.

[0033] For certain instructions, the successful completion of an instruction includes the vectoring of a fault, exception or interrupt. For example, in the IA-32 ISA, a software interrupt instruction (i.e., INTn, INTO, INT3) vectors a software interrupt by generating a call to an interrupt or exception handler. When such an instruction takes place, the VM exit due to the single stepping mechanism occurs prior to execution of the first instruction of the interrupt or exception handler in the VM (assuming there are no nested exceptions encountered during delivery of the software interrupt and no VM exits are caused by the faults or exceptions).

[0034] Note that the VM may be utilizing debug or single stepping mechanism independent of the single stepping mechanism discussed herein. For example, in the IA-32 ISA, the VM may have the TF bit in the EFLAGS register set to one in order to generate single stepping traps, or it may be utilizing the debug registers to generate debug breakpoints. If the first VM instruction causes such a breakpoint or trap, the VM exit due to the single stepping mechanism occurs before the vectoring of the breakpoint or trap. Information regarding the pending breakpoint or trap may, in an embodiment, be stored in the VMCS for use by the VMM to properly emulate the mechanism used by the VM.

[0035] **Figure 2** is a flow diagram of one embodiment of a process 200 for providing support for single stepping a VM in a VM environment. The

process may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, programmable logic, microcode, etc.), software (such as that run on a general purpose computer system or a dedicated machine), or a combination of both. In one embodiment, process 200 is performed by single stepping logic 122 of **Figure 1**.

[0036] Referring to **Figure 2**, process 200 begins with processing logic receiving a request to transition control to a VM from a VMM (processing block 202). In one embodiment, the request to transition control is received via a VM entry instruction executed by the VMM.

[0037] At decision box 204, processing logic determines whether the VMM has requested single stepping of the VM that is to be invoked. In one embodiment, processing logic determines whether the VMM has requested single stepping of the VM by reading the current value of a single stepping indicator maintained by the VMM. The single stepping indicator may reside in the VMCS or any other data structure accessible to the VMM and processing logic 200. In one embodiment, when the VMM wants to single step a VM, the VMM sets the single stepping indicator to a single stepping value (e.g., 1) and then generates a request to transfer control to this VM. In one embodiment, if the VMM sets the single stepping indicator to the single stepping value, it also sets execution control indicators to cause a VM exit on each fault which may occur during the execution of the VM. Alternatively, the setting of the single stepping indicator is independent of some or all execution control indicators.

[0038] In one embodiment, the single stepping indicator is reset to a non-single stepping value (e.g., 0) at each VM exit. In another embodiment, the VMM resets the single stepping indicator to the non-single stepping value prior to requesting the transfer of control to a VM if no single stepping of the VM is needed.

[0039] As discussed above, the VMM may request single stepping of a VM when, for example, debugging system or application code running in the VM, virtualizing a device or platform during the operation of the VM, tracing the execution of the VM, etc.

[0040] If processing logic determines that single stepping of the VM is needed, processing logic executes a first instruction in the VM (processing block 208) and determines whether the execution of the first VM instruction completes successfully (decision box 210). If so, processing logic generates a VM exit (processing block 212) and informs the VMM that the reason for the VM exit is single stepping (processing block 214). In one embodiment, processing logic informs the VMM about this reason by generating a designated reason code. In one embodiment, the reason code is stored in the VMCS.

[0041] If the execution of the first VM instruction is unsuccessful, processing logic determines whether the resulting fault is to cause a VM exit (decision box 216). This determination may be made by examining relevant execution control information (e.g., stored in the VMCS 124). If the resulting fault is to cause a VM exit, processing logic generates a VM exit (processing

block 222) and informs the VMM that the reason for the VM exit is the fault (processing block 224). If the resulting fault does not cause a VM exit, processing logic delivers the fault to the VM (processing block 218) and determines whether the delivery of the fault completes successfully (decision box 220). If the delivery of the fault completes successfully, processing logic generates a VM exit (processing block 212) prior to executing any instructions of a corresponding fault handler and informs the VMM that the reason for the VM exit is single stepping (processing block 214).

[0042] If the delivery of the fault is unsuccessful, processing logic returns to decision box 216 to process the resulting new fault.

[0043] If processing logic determines at decision box 204 that single stepping of the VM has not been requested, processing logic executes a first instruction in the VM (processing block 226) and determines whether the execution of the first VM instruction completes successfully (decision box 228). If so, processing logic executes the next instruction in the VM (processing block 230) and returns to decision box 228. Otherwise, if the execution of the first VM instruction is unsuccessful, processing logic determines whether the resulting fault is to cause a VM exit (decision box 232). If so, processing logic generates a VM exit (processing block 222) and informs the VMM that the reason for the VM exit is the fault (processing block 224). If not, processing logic delivers the fault to the VM (processing block 234) and determines whether the delivery of the fault completes successfully (decision box 236). If the delivery of the fault completes successfully,

processing logic execute a first instruction of the fault handler and returns to decision box 228. If the deliver of the fault is unsuccessful, processing logic returns to processing block 234 to process the resulting new fault.

[0044] In some embodiments, certain events may occur during the operation of a VM invoked in response to a request for a VM entry associated with a single stepping indicator set to a single stepping value. For example, the VMM may require that a fault be delivered to a VM as part of a VM entry, a VM entry may put the processor in a non-active activity state, a first instruction following a VM entry may cause a VM exit, etc. In the instruction set architecture (ISA) of the Intel® Pentium® 4 (referred to herein as the IA-32 ISA), examples of non-active activity state include the Halt, Wait-for-SIPI, shutdown, and MWAIT states. While in these non-active activity states, the processor does not complete any instructions, and instead waits for the occurrence of one or more break events, which may move the processor from the non-active activity state to the normal active state. For example, while in the Halt activity state, the processor waits for the occurrence of unblocked hardware interrupts, system management interrupts (SMIs), system initialization messages (INITs), non-maskable interrupts (NMIs), debug exceptions, machine check exceptions, etc. Exemplary processes handling such events while supporting single stepping of a VM are discussed below in conjunction with **Figures 3 and 4**.

[0045] **Figure 3** is a flow diagram of one embodiment of a process 300 for responding to a request for a VM entry with a single stepping indicator set

to a single stepping value. The process may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, programmable logic, microcode, etc.), software (such as that run on a general purpose computer system or a dedicated machine), or a combination of both. In one embodiment, process 300 is performed by single stepping logic 122 of **Figure 1**.

[0046] Referring to **Figure 3**, process 300 begins with processing logic detecting a VM entry with a single stepping indicator set to a single stepping value (processing block 302). Next, at decision box 304, processing logic determines whether the VMM has requested a delivery of a pending fault to the VM as part of the VM entry (i.e., a vectored fault is to be delivered to the VM). If so, processing logic delivers the pending fault to the VM (processing block 306). Delivery of a fault may involve searching a redirection structure for an entry associated with the fault being delivered, extracting from this entry a descriptor of the location of a corresponding fault handler, and jumping to the beginning of the fault handler using the descriptor. During the delivery of a fault, several operations may need to be performed (e.g., address translations, validation of segmentation registers and access addresses, permission level checks, etc.) that may result in new faults (e.g., page faults, general-protection faults, etc.). At decision box 308, processing logic determines whether the delivery of the pending fault completes successfully. If so (i.e., no new faults occur), processing logic generates a VM exit with the

reason code specifying the single stepping indicator, prior to executing any instructions in the corresponding handler (processing block 310).

[0047] If the delivery of the pending fault is unsuccessful, processing logic determines whether a resulting new fault is to cause a VM exit (decision box 312). If so, processing logic generates a VM exit with the reason code specifying the new fault (processing block 314). Otherwise, if the new fault is not to cause a VM exit, processing logic returns to processing block 306 to process the new fault.

[0048] If processing logic determines at decision box 304 that the VMM has not requested delivery of a pending fault as part of the VM entry, processing logic further determines whether the VM entry is to put the processor in a non-active activity state (decision box 316). If so, processing logic enters the non-active activity state (e.g., the Halt state) (processing block 317), waits for a break event that can put the processor from the non-active activity state to a normal (i.e., active) activity state (processing block 318), and then determines whether the break event is to cause a VM exit (e.g., by examining relevant execution control information in the VMCS 124) (decision box 320). If this determination is positive, processing logic generates a VM exit with the reason code specifying the break event (processing block 322). In an embodiment, if a specifier of the activity state of the VM is required to be saved as part of the VM exit (as determined, for example, by examining a save activity state indicator in the execution control information in the VMCS), the specifier of the activity state prior to the occurrence of the break event (i.e., the

non-active activity state) is saved. Otherwise, if the break event is not to cause a VM exit, processing logic proceeds to processing block 306, attempting to deliver the break event to the VM. In one embodiment, the attempt to deliver the break event to the VM has the effect of moving the VM from the non-active activity state to the normal (i.e., active) activity state; if a specifier of the activity state of the VM is subsequently saved, it will indicate the normal (i.e., active) activity state.

[0049] If processing logic determines at decision box 316 that the VM entry does not put the processor in a non-active activity state, processing logic executes the first instruction in the VM (processing block 324) and determines whether the execution of the first instruction is to cause a VM exit (decision box 326). If so, processing logic generates a VM exit with the reason code identifying the instruction (processing block 328). If not, processing logic determines whether execution of the first instruction completes successfully (decision box 330).

[0050] If the execution of the first instruction is unsuccessful, processing logic proceeds to decision box 312 to process a resulting new fault. If the execution of the first instruction completes successfully, processing logic further determines whether the instruction causes the processor to go into a non-active activity state (decision box 332). For example, in the IA-32 ISA, the HLT instruction causes the processor to enter the Halt state, which is a non-active activity state. If the executed instruction does enter a non-active activity state, processing logic further determines whether a specifier of the

activity state is required to be saved (e.g., by examining a save activity state indicator in the execution control information in the VMCS) (decision box 334). If the specifier of activity state is required to be saved, processing logic saves the indicator of the activity state (e.g., into the VMCS) (processing block 336) and generates a VM exit with the reason code specifying the single stepping indicator (processing block 338). If the instruction does not cause the processor to go into a non-active activity state, or it does causes the processor to go into a non-active activity state but the specifier of the activity state does not need to be saved, processing logic proceeds directly to processing block 338.

[0051] In some embodiments, single stepping may have a priority that is below the priority of some other events (e.g., platform events such as, for example, INIT and SMI). **Figure 4** is a flow diagram of one embodiment of a process 400 for handling events having a higher priority than single stepping. The process may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, programmable logic, microcode, etc.), software (such as that run on a general purpose computer system or a dedicated machine), or a combination of both. In one embodiment, process 400 is performed by single stepping logic 122 of **Figure 1**.

[0052] Referring to **Figure 4**, process 400 begins with processing logic detecting a VM entry with a single stepping indicator set to a single stepping value (processing block 402) and executes a first instruction in the VM (processing block 404).

[0053] Next, following execution of the first instruction in the VM, processing logic detects an event (e.g., NMI, SMI, etc.) that has a higher priority than single stepping (processing block 406) and determines whether the higher priority event is to be handled by the VMM or some designated software (decision box 408). In one embodiment, the higher priority event is handled by the VMM if the VMM utilizing the single stepping mechanism was capable of setting, prior to the relevant VM entry, an execution control indicator for the higher priority event in such a way as to cause a VM exit. If the VMM utilizing the single stepping mechanism was not capable of setting a corresponding execution control indicator as discussed above (e.g., there is no such an execution control indicator available), the higher priority event may be handled by designated software. For example, the VMM utilizing the single stepping mechanism may set, prior to the relevant VM entry, corresponding execution control bits such that an NMI will cause a VM exit. In another example, the VMM utilizing the single stepping mechanism may not have the capability of updating an execution control indicator associated with SMI prior to the relevant VM entry. Then, SMI may be handled by software designated to manage higher priority events.

[0054] If the higher priority event is to be handled by the VMM utilizing the single stepping mechanism, processing logic generates a VM exit with the reason code specifying the higher priority event (processing block 410). Alternatively, if the higher priority event is to be handled by the software designated to manage the higher priority event, processing logic sets

a pending VM exit indicator (e.g., in the VMCS) to a single stepping value (processing block 412), and transfers control to the designated software (processing block 414). The designated software then handles the higher priority event and facilitates the delivery of the pending VM exit due to the single stepping mechanism to the VMM (block 416). The pending VM exit indicator may be delivered to the VMM by the designated software itself or by the processor upon completion of the processing of the higher priority event. For example, SMI processing may be handled by managing software separate from the VMM. Upon completion of handling an SMI, the managing software may execute the RSM (return-from-system-management-mode) instruction, and then, upon execution of this instruction, the processor may recognize that there is a pending VM exit due to the single stepping mechanism and deliver it accordingly.

[0055] As discussed above, single stepping of a VM by the VMM may have various uses, including, for example, debugging system or application code running in the VM, virtualizing a device or platform during the operation of the VM, tracing the execution of the VM, etc. An exemplary process utilizing single stepping to facilitate virtualization of a device will now be discussed in more detail.

[0056] **Figure 5** is a flow diagram of an exemplary process 500 for utilizing single stepping of a VM to facilitate virtualization of a device, according to one embodiment. The device may be an input/output (I/O) device that uses a memory-mapped programming model (e.g., an advanced

programmable interrupt controller (APIC)), and the VMM may want to prevent a VM from writing data to the memory-mapped I/O device while allowing the VM to read data from the memory-mapped I/O device. Process 500 may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, programmable logic, microcode, etc.), software (such as that run on a general purpose computer system or a dedicated machine), or a combination of both.

[0057] Referring to **Figure 5**, process 500 begins at processing block 502 with processing logic in the VMM setting page tables of the VM to make the memory-mapped space of the device non-writable and configuring page faults encountered during the execution of the VM to cause VM exits (e.g., using corresponding execution control information). In one embodiment, setting of the VM page tables is enabled by using a virtual translation lookaside buffer (VTLB) data structure that maintains a separate active guest page table, under control of the VMM, for each VM.

[0058] At processing block 504, processing logic in the processor generates a page fault VM exit when the VM accesses the memory-mapped space of the device.

[0059] At processing block 506, processing logic in the VMM determines that the VM access was to the protected memory-mapped space (e.g., by examining the faulting address value that falls onto the memory-mapped I/O page).

[0060] At processing block 508, processing logic in the VMM changes the VM page tables to map the faulting address to a different physical memory page located in normal physical memory (i.e., not memory mapped I/O space).

[0061] At processing block 510, processing logic in the VMM sets a single stepping indicator and requests a VM entry.

[0062] At processing block 512, processing logic in the processor executes an instruction in the VM to access the above page chosen by the VMM.

[0063] At processing block 514, processing logic in the processor causes a VM exit, following a successful execution of the instruction, due to the single stepping indicator.

[0064] Next, processing logic in the VMM examines data written to the page of physical memory configured in processing block 508 (processing block 516) and decides whether the data is to be written to the device (decision box 518). This decision may be based on security (e.g., it may be allowed to write some data to the device but not the other data) or correctness (e.g., the data may need to be changed prior to writing it to the device if required by virtualization).

[0065] If the decision made at decision box 518 is positive, processing logic in the VMM writes the data to the device (processing block 520). Alternatively, the VMM may modify the data before writing it to the device to

ensure that certain characteristics regarding, for example, security, performance or device virtualization are proper.

[0066] Thus, a method and apparatus for providing support for single stepping a virtual machine in a virtual machine environment have been described. It is to be understood that the above description is intended to be illustrative, and not restrictive. Many other embodiments will be apparent to those of skill in the art upon reading and understanding the above description. The scope of the invention should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.